Table Of Contents

Adding the RedLaser SDK to your Project

The Developer SDK for RedLaser includes a Sample app that shows how the SDK can be used. It is intended to be a simple demonstration of the SDK's capabilities, and shows a somewhat simple way to use the SDK from an app.

The source code in the Sample app itself should not be confused with the SDK. You are free to copy the code in the Sample app if you wish, but you're also free to use the SDK in a way different than what the Sample app does.

Required Files

To add the SDK to your app's project, you need to add 2 files to your target:

* RedLaserSDK.h
* libRedLaserSDK.a

If you are a licensee, and don't want to use the SDK in evaluation mode, you'll also need to add:

* RedLaser_License.xml

The "RedLaser_License.xml" file needs to be generated for your app from the redlaser.com website. See the section on licensing for details.

Optional Files

The SDK package contains a set of image and sound assets that your application may use as part of the app-supplied overlay. These files should make it easier to create an overlay similar to the one used by the RedLaser app. The SDK itself does not use these files.

As a separate issue, showing the "powered_by_redlaser.png" image to the user when scanning is generally a requirement of the SDK license; from a technical standpoint, you can rename the image file, or make it a .jpg, or incorporate it into a larger overlay graphic. The SDK doesn't attempt to access it.

Frameworks

The RedLaser library itself depends on the following Apple frameworks:

- UIKit.framework
- Foundation.framework
- QuartzCore.framework
- CoreVideo.framework

- CoreMedia.framework
- CoreGraphics.framework
- AVFoundation.framework
- OpenGLES.framework
- Security.framework
- libiconv.dylib
- libstdc++.dylib

Depending on how you set up your project, the standard c++ library may or not be included by default.

Linking

The minimum OS version for the RedLaser SDK is 4.0. Earlier versions of the SDK had a lower minimum OS version, but when Apple released the AVFoundation framework with iOS 4.0, it became the only approved way to access the camera.

Applications that wish to run on iOS versions prior to 4.0 and still use RedLaser features on 4.0 and above can still do so. Developers should be able to weak link against the appropriate OS frameworks, and have their apps load on iOS 3.x. We've provided a convenience function, RL_CheckReadyStatus(), that checks whether the frameworks required by RedLaser are available. If this function returns RLState_MissingOSLibraries, your app should not attempt to instantiate an instance of BarcodePickerController. It will almost certainly crash your app.

Hardware Considerations

The RL_CheckReadyStatus function also checks for the existence of a camera on the device. If no camera is found, this function returns RLState_NoCamera, and you shouldn't use RedLaser (although it won't crash if you attempt it, it doesn't do anything useful).

The SDK can be used in the iPhone Simulator, but cannot scan actual barcodes. Even if your computer has a webcam, it isn't hooked up to the simulator's AVFoundation framework. When running in the Simulator, the SDK's view will show a UIButton that 'fakes' recognizing a barcode when clicked.

Licensing

The RedLaser SDK can be used with or without a license file. With no license file, the SDK runs in evaluation mode, and limits the number of scans per device. The limitations imposed when a license file is present depend on your licensing terms.

When you signed up as a developer, you generated a license file named "RedLaser_License.xml". You will need add this file to your project.

The license file is digitally signed. Do not modify it. This includes modifying line endings, which some source control systems like to do. The files are currently being generated with Unix-style LF line endings, although it's possible that could change; the correct line ending style is 'what it was signed with'.

Your license file has a list of all the application bundle IDs you've registered with RedLaser.com. The SDK will only work correctly in licensed mode if it's being used in one of the applications you've registered.

Using the RedLaser SDK in your App

This section goes through the "RedLaserSDK.h" header file one section at a time, explaining each section as we go.

Before You Use the SDK

The RedLaser SDK requires an iOS device with a camera. It requires iOS version 4.0. It has certain licensing requirements, and will refuse to run if these aren't met.

Before using the SDK, your app should call RL_CheckReadyStatus(). If this function returns a negative value, your app should not attempt to use the SDK (most likely, your app should disable the button that opens the BarcodePickerController).

The SDK will show alerts to the user if it's called but cannot scan--for example, if it's opened on a device that has no camera. These alerts, however, are a fallback position--they're not localized, and may not provide the user with the best experience. Therefore, it's better for your app to use this function and take appropriate action beforehand.

The value returned from this function could change within an application run if the underlying state changes--for example, if you're using the SDK in evaluation mode and you hit the maximum scan count for a device.

The MissingOSLibraries result should only occur if you've weak-linked iOS libraries that RedLaser needs to function, and those libraries aren't available at run time. All the iOS libraries that RedLaser currently needs are available in iOS version 4.0 and later. If your app needs to run on OS versions prior to 4.0, you should weak link the OS libraries used by RedLaser and call RL_CheckReadyStatus() before trying to use RedLaser.

Supported Barcode Types

```
#define kBarcodeTypeEAN13 0x1
#define kBarcodeTypeUPCE 0x2
```

```
#define kBarcodeTypeEAN8 0x4
#define kBarcodeTypeSTICKY 0x8
#define kBarcodeTypeQRCODE 0x10
#define kBarcodeTypeCODE128 0x20
#define kBarcodeTypeCODE39 0x40
#define kBarcodeTypeDATAMATRIX 0x80
#define kBarcodeTypeITF 0x100
#define kBarcodeTypeEAN5 0x200
#define kBarcodeTypeEAN2 0x400
```

If you're new to working with barcodes, there's a good chance you're looking to scan UPC codes, and are about to choose UPCE. You probably want to look at EAN13 instead. EAN13 is a newer standard that is a strict superset of the UPC-A standard. You might want to scan UPC-E barcodes as well, just realize that UPC-E barcodes are the 6-digit shortened barcodes found on some products, not the 12 digit codes which are more common.

Although DATAMATRIX is in the list, it doesn't work very well (yet) and is not officially supported at this time.

EAN5 and EAN2 are codes that can appear next to an EAN13 code, and specify pricing information or issue numbers for certain products. These barcodes are 'associated codes' that are only searched for when a EAN13 is found. If you enable EAN13 or EAN8, you can enable EAN5 and/or EAN2 as well, and information about the associated codes will be returned to you. Enabling only EAN5 or EAN2 will not produce any results.

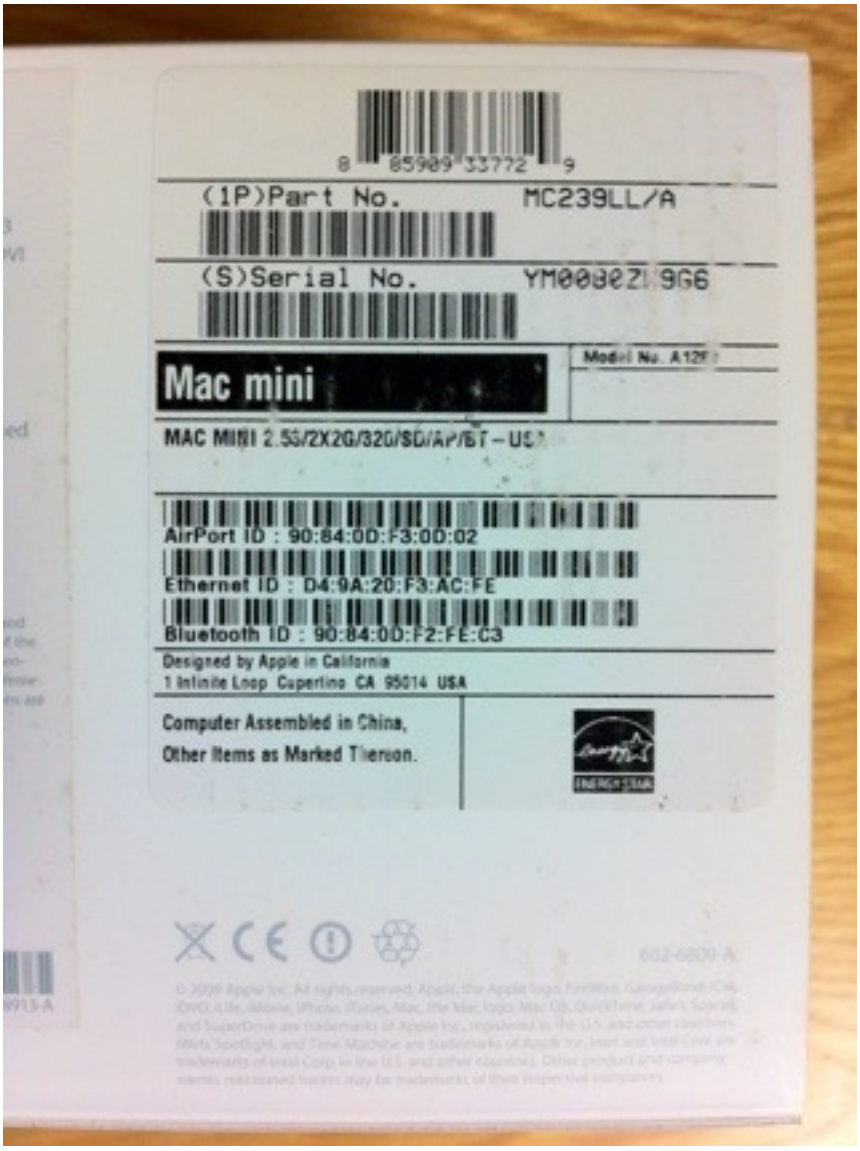
For best performance scanning StickyBits barcodes, we recommend disabling kBarcodeTypeEAN13, particularly if using a non-focusing camera.

BarcodeResult class

```
/**************************************************************************
  BarcodeResult

  The return type of the recognizer is a NSSet of Barcode objects.
*/

@interface BarcodeResult : NSObject { }

@property (readonly) int barcodeType;
@property (readonly) NSString *barcodeString;
@property (readonly, copy) NSString *extendedBarcodeString;
@property (readonly) BarcodeResult *associatedBarcode;
```

```
@property (readonly, retain) NSDate *firstScanTime;
@property (readonly, retain) NSDate *mostRecentScanTime;
@property (readonly, retain) NSMutableArray *barcodeLocation;
@end
```

The preferred return method for SDK version 3.0 delivers a NSSet of these objects, one for each barcode that was recognized during the scanning session. Why does the SDK do this? To handle situations like this:



The barcodeType property will be one of the values from the list above, and barcodeString will be the value read from the barcode. ExtendedBarcodeString, if non-nil, contains extra information about the barcode. Currently, this field is only used for UPC-E barcodes, to give their UPC-A equivalent.

In the case where the SDK produced a EAN with an associated EAN5 or EAN2 code, both codes are included as separate BarcodeResults in the returned result set, and each of them will have their associatedBarcode set to the other.

Each barcode in the set will have 2 NSDates: the first time it was recognized during the session, and the last time it was recognized during the session.

Finally, each barcode will have a NSArray of NSValues, where each NSValue is a CGPoint, indicating where we located the barcode. The coordinates of the points will be in the same coordinate system as the BarcodePickerController's bounds. The first point in the array will be the top left of the barcode, and the second will be the top right of the barcode. Note that if a barcode is recognized 'upside down', these points will be in the lower right and lower left when viewed onscreen. Also, because the preview is mirrored when using a device's front camera for recognizing, the points aren't necessarily in clockwise winding order either. The array will usually contain 4 points, but it could contain more or fewer.

The path produced from these points may not cover the entire barcode, and may be only one pixel high or wide. The barcode location is only updated on frames where the barcode is actually recognized, so the longer it's been since mostRecentScanTime, the less likely it is that the barcode is still at that position in the camera preview. Barcodes recognized by the partial recognition method (used for some long barcodes, allowing the user to point the camera at each part of the barcode and piece the full code together) will only have recognition information on the most recent part of the barcode to be scanned.

FindBarcodesInUIImage

This method analyses a given image and returns information on any barcodes discovered in the image. It is intended to be used in cases where the user already has a picture of a barcode (in their photos library, for example) that they want to decode. This method performs a thorough check for all barcode symbologies we support, and is not intended for real-time use.

When scanning barcodes using this method, you cannot (and need not) specify a scan orientation or active scan region; the entire image is scanned in all orientations. Nor can you restrict the scan to particular symbol types. If such a feature is absolutely necessary, you can implement it post-scan by filtering the result set.

FindBarcodesInUIImage operates synchronously, but can be placed in a thread. Depending on image size and processor speed, it can take several seconds to process an image.

BarcodePickerControllerDelegate

```
/*****************************************************************************
   BarcodePickerControllerDelegate

   The delegate receives messages about the results of a scan.
   barcodePickerController:didScanBarcode:withInfo: is the legacy API;
   barcodePickerController:returnResults: is the new API that supports multiple
   barcode capture.
*/

@protocol BarcodePickerControllerDelegate <NSObject>
@optional

- (void) barcodePickerController:(BarcodePickerController*)picker
      returnResults:(NSSet *)results;
@end
```

This is the delegate object of the SDK that receives the results of a scan session after the scan session completes.

Your application should always check for multiple barcodes returned from a scan session. It is possible for multiple codes to get recognized at the same time if multiple barcodes are visible in the preview frame. You can use the activeRegion property of the BarcodePickerController to restrict the scanning area if necessary, and can also inspect the onscreen locations of the scanned barcodes if you need to determine which barcode the user intended to scan.

If implemented in the delegate, the SDK will always call barcodePickerController:returnResults: exactly once per scan session. This will happen after doneScanning is called, or if the session encounters an error. This means that the delegate may be called with an empty set of barcodes.

CameraOverlayViewController

```
/*****************************************************************************
   CameraOverlayViewController

   An optional overlay view that is placed on top of the camera view.
   This view controller receives status updates about the scanning state, and
   can update the user interface.
*/

@interface CameraOverlayViewController : UIViewController { }
@property (readonly, retain) BarcodePickerController *parentPicker;
- (void)barcodePickerController:(BarcodePickerController*)picker statusUpdated:(NSDictionary*)status;
```

```
@end
```

Your application should create a custom subclass of CameraOverlayViewController, and set the overlay property of the BarcodePickerController to your custom overlay before displaying the BarcodePickerController.

If you don't set a custom overlay, the default overlay is used, which has no user interface and simply exits the scan session as soon as the first barcode is recognized. With no UI, there is no way to cancel, so we really do recommend that you implement an overlay view.

Other than normal UIViewController operations, your overlay receives barcodePickerController:statusUpdated messages from the SDK. The status NSDictionary may contain these keys when this method is called:

Key - ValueType	Usage Notes
@”FoundBarcodes” - NSSet	NSSet of BarcodeResult objects. Will be the empty set if no barcodes have been found yet.
@”NewFoundBarcodes” - NSSet	NSSet of BarcodeResult objects. This key only exists in the NSDictionary if at least one new barcode was recognized in the most recent scan pass.
@”Valid” - NSNumber	TRUE if there are valid results.
@”InRange” - NSNumber	TRUE if there is a barcode in range in the viewfinder. This key may be used to advise the user to hold the phone steady while a barcode is read. Not all devices require this guidance in order to successfully read barcodes--be sure to test on non-focusing cameras to see this key in use.
@”Guidance” - NSNumber	This key only exists in the NSDictionary if there is guidance to be given. 1 means that the SDK sees a likely barcode in range, but hasn't been able to decode it for several seconds. The overlay may use this to advise the user to try scanning the barcode in parts by holding the phone close to each part of the barcode. 2 means that the SDK has scanned the first part of a barcode, the contents of which are available in the @"PartialBarcode" property. The overlay may use this to advise the user of the part of the barcode that's been successfully scanned.

Key - ValueType	Usage Notes
@”PartialBarcode” - NSString	The part of a barcode that's been scanned when doing partial scanning. The overlay may show the partially scanned barcode text to the user to help guide them on completing the partial scan.
@”CameraSupportsTapToFocus” - NSNumber	TRUE if the device supports tap to focus. The overlay should only guide the user to tap on the barcode to focus on it if tap to focus is supported. Note that this guidance isn’t mandatory; recognition will happen without it.

The overlay is responsible for determining when to end a scanning session. It can do this by implementing a "Done" or "Cancel" button, or by exiting when a condition is met in the status callback--such as the Valid key being TRUE. Either way, the overlay should signal the SDK that it is time to end the scan session by calling the BarcodePickerController's doneScanning method.

The status method will always be called from the main thread.

BarcodePickerController

```
/**************************************************************************
   BarcodePickerController

   This ViewController subclass runs the RedLaser scanner, detects barcodes, and
   notifies its delegate of what it found.
*/
@interface BarcodePickerController : UIViewController { }

- (void) pauseScanning;
- (void) resumeScanning;
- (void) doneScanning;
- (BOOL) hasFlash;
- (void) turnFlash:(BOOL)value;

@property (nonatomic, retain) CameraOverlayViewController *overlay;
@property (nonatomic, assign) id <BarcodePickerControllerDelegate> delegate;
@property (nonatomic, assign) BOOL scanUPCE;
@property (nonatomic, assign) BOOL scanEAN8;
@property (nonatomic, assign) BOOL scanEAN13;
@property (nonatomic, assign) BOOL scanSTICKY;
@property (nonatomic, assign) BOOL scanQRCODE;
@property (nonatomic, assign) BOOL scanCODE128;
```

```
@property (nonatomic, assign) BOOL scanCODE39;
@property (nonatomic, assign) BOOL scanDATAMATRIX;
@property (nonatomic, assign) BOOL scanITF;
@property (nonatomic, assign) BOOL scanEAN5;
@property (nonatomic, assign) BOOL scanEAN2;
@property (nonatomic, assign) CGRect activeRegion;
@property (nonatomic, assign) UIImageOrientation orientation;
@property (nonatomic, assign) BOOL torchState;
@property (readonly, assign)  BOOL isFocusing;
@property (nonatomic, assign) BOOL useFrontCamera;

@end
```

In order to scan barcodes, the application should create an instance of BarcodePickerController, configure its properties, and then display it to the user. Here's some example code that does this:

```
- (void) startScan
{
        BarcodePickerController *picker = [[BarcodePickerController alloc] init];
        [picker setOverlay:customOverlay];
        [picker setDelegate:self];

        // Initialize with portrait mode as default
        picker.orientation = UIImageOrientationUp;

        // hide the status bar
        [[UIApplication sharedApplication] setStatusBarHidden:YES];

        // Show the scanner
        [self presentModalViewController:picker animated:TRUE];
        [picker release];
}
```

Note that your application is not limited to using presentModalViewController:animated: to display the picker view. Your application is, however, generally responsible for including some method of ending a scanning session by adding appropriate UI to the overlay.

The delegate and overlay properties should be set up before initiating scanning or displaying the BarcodePickerController. It *may* work to change them during scanning, but it is not a supported use case. If you really need this functionality, please give us feedback.

The other listed properties may be modified both before and during scanning. The scan* properties may be set to choose what barcode types the application is interested in. Setting the orientation property to UIImageOrientationUp will scan in portrait mode,

while UIImageOrientationLeft or UIImageOrientationRight will scan in landscape mode. The activeRegion property is used to restrict the scan area to a subset of the overall view. TorchState is used to turn the LED torch on an iPhone 4 on or off.

The useFrontCamera property acts as a recommendation if TRUE; devices that do not have a front facing camera will still use the default camera.

PauseScanning and resumeScanning may be used to temporarily halt image processing.

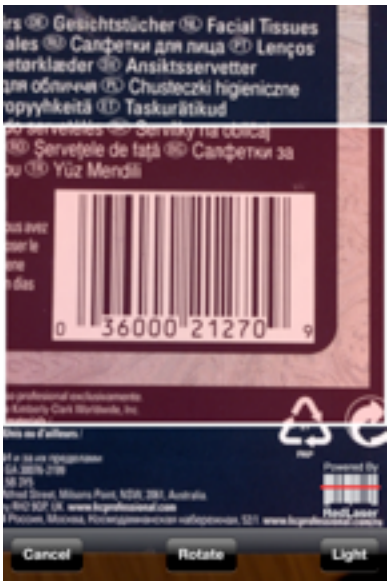
DoneScanning is used to end a scan session. Scan sessions do not end by themselves when a barcode is recognized; if you want this behavior you need to call doneScanning when barcodePickerController:StatusUpdated: indicates that a code has been recognized.

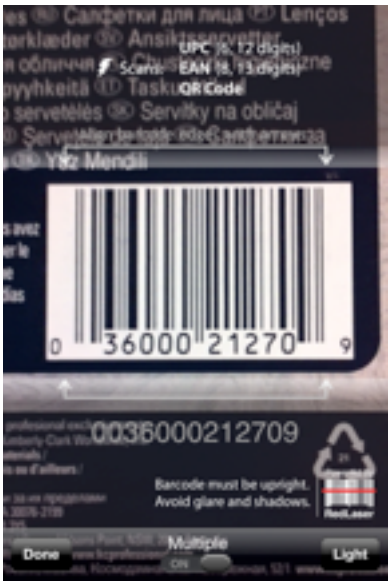
Designing Your Overlay

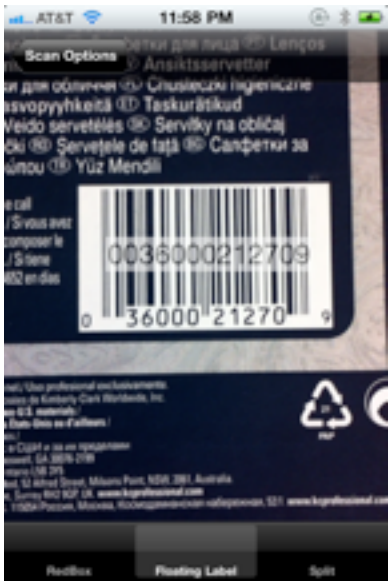
The CameraOverlayViewController is a subclass of UIViewController. You should design a subclass of CameraOverlayViewController in your app, and set the overlay property of the BarcodePickerController to the overlay you create before showing the BarcodePickerController.

The CameraOverlayViewController should implement the barcodePickerController:statusUpdated: method, to receive regular status updates from the SDK. You can use these status updates to make your UI responsive to the state of the scan session.

The Sample app (left image, below) has an overlay that contains a toolbar with 3 buttons, and a red shaded area guiding the user where best to place the barcode. The RedLaser app in the app store (middle image) has a different overlay, with arrows that indicate optimal placement, and change color when the barcode is in range (that is when @”InRange” is true). With the information the SDK delivers to your overlay, other user interfaces are possible as well.







Capturing Barcodes in Sections

The 3.0 SDK introduces a new method of scanning barcodes, where the user can scan a barcode in parts and the SDK will stitch the parts together to recognize the full barcode. This scan method is useful for capturing long barcodes on older devices, where the camera resolution isn't high enough to resolve the entire barcode at once.

Currently, this new method only works for Code 39 barcodes.

The algorithm to stitch partial barcodes is always enabled in the SDK, however without app-level support in the overlay to guide the user, it is very unlikely a user will ever find it. Most of this document is a discussion of how to design your overlay UI to guide a user through the process of scanning a barcode in parts. Note that all of the cues from the SDK relating to this feature can be safely ignored if you don't want to support partial barcode scanning–you don't **have** to do anything.

Many SDK clients aren't using Code 39 in their apps, and even some of those that are may not need this feature. For developers that need Code 39 support in their app, adding this optional feature can lead to a better user experience, particularly for users with older devices trying to scan Code 39 codes in excess of ~15 characters in length.

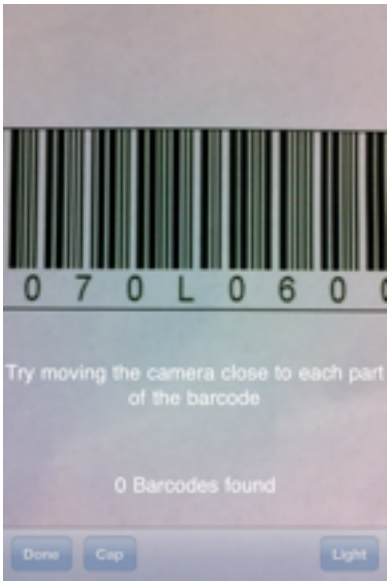
The basic interaction for partial barcode scanning is that the SDK tells the overlay what it sees, and the overlay tells the user what to do to successfully complete the scan. The communication from the SDK to the overlay happens through the @"Guidance" and @"PartialBarcode" keys.

Guidance == 1

After a couple of seconds where the SDK sees that the user is pointing the camera at something that looks like a barcode, if the SDK has not recognized the barcode yet it will start returning 1 in the “Guidance” key of the barcodePickerController:statusUpdated: dictionary.

Note that at this point the SDK isn’t claiming to know for sure if it’s looking at a Code 39 barcode, or for that matter any kind of barcode; it is just looking at something vaguely barcode-like that the SDK can’t fully recognize.

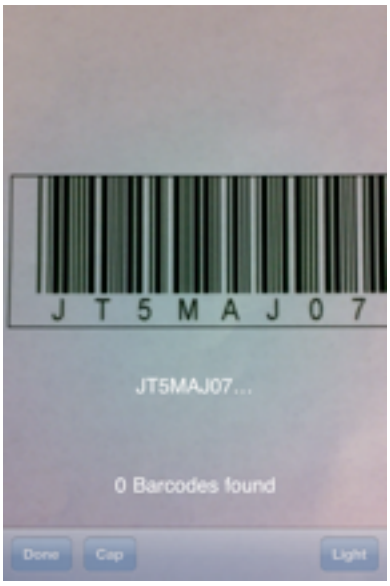
When the Guidance key goes to 1, the overlay should advise the user to try scanning the barcode in parts by holding the phone close to the first part of the barcode, and then scanning across slowly. See the image below.



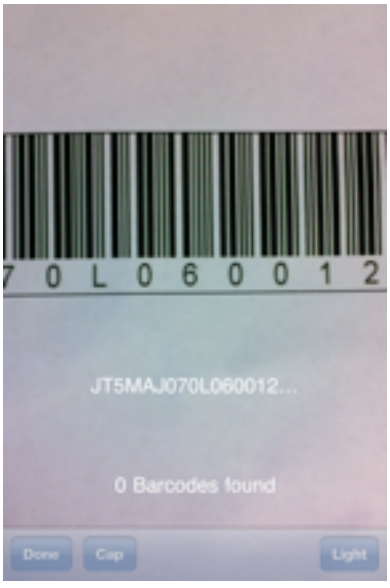
Once the user does this, the first part of the barcode may be scanned, at which time the Guidance key changes to 2.

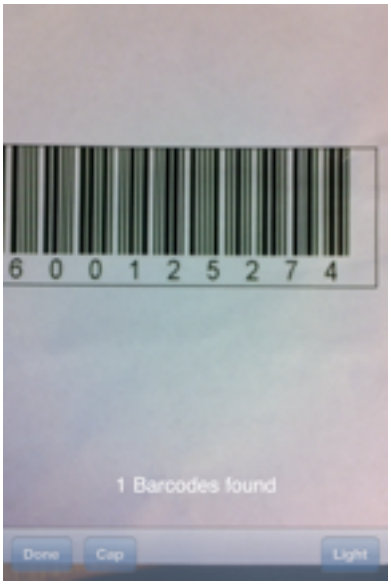
Guidance == 2

When the guidance changes to 2, an additional key is available, “PartialBarcode”. This key contains the text of the barcode that has so far been decoded. In the example shown below, the partial barcode contains “JT5MAJ07” (the ellipsis is added by the overlay in this example).



As the user scans their device across the barcode, characters will be added to the partial barcode, as shown in the image below left. When they reach the end of the barcode, the completed code is added to the set of found barcodes, and will be sent to the app's CameraOverlayViewController subclass as an element in the "FoundBarcodes" set, just as if the barcode was recognized normally, as shown in the right image below.





Once a barcode has been fully scanned, the Guidance key goes back to zero and the PartialBarcode key will no longer be sent to the overlay during that scan session.

UI Notes

The intent of this design is to allow the app developer significant leeway in how they want to enable this feature for the user. Apps are free to ignore it altogether, or they can provide guidance in the user interface as they see fit. The sample images below show various text strings being shown by the overlay, but what the overlay shows the user is entirely up to the app developer.

Other than the text of the barcode string itself, localization for this feature is up to the app as all the strings come from the app. Or, an app developer could use an icon or a small animation to inform the user as to what to do to scan using this method.

Notes on Specific Barcode Types

This section contains supplemental information about how the SDK returns data for specific barcode types.

UPC-A and EAN-13

The SDK does not have a type for UPC-A, as it has been superseded by EAN-13. When the SDK encounters a UPC-A barcode, it will return the equivalent EAN-13 number, which prepends a 0 to the UPC code. The reasoning for why we do this has a lot to do with the fact that EAN-13 codes whose first number is 0 look exactly the same (to a barcode reader) as UPC-A codes.

UPC-E

UPC-E codes are shortened codes that use the same number system as UPC-A (and EAN-13). There is a specific algorithm for expanding a UPC-E code's digits and recovering the full EAN-13 code. The extendedBarcodeString member of the BarcodeResult object will contain the full EAN-13 code in this case.

QR Codes

When a QR Code is detected, the SDK returns information about the QR Code's location such that the top left and top right corners are oriented as shown in the image below.

Code 128

The SDK inserts the string “]C1” when the first symbol in a Code 128 barcode is FNC1, and inserts ASCII code 29 (the Group Separator character) when the FNC1 symbol is found elsewhere in the string.

FNC codes 2 through 4 are ignored and not inserted into the output string.

ZXing License Attribution